PARCEL v1.04
User Guide

Lawrence C. Cowsar[*] Carol A. San Soucie[†] Ivan Yotov[†]

May, 1996

1 Introduction

This document is provided to describe the calling sequence and to augment the internal documentation of **PARCEL**, a parallel cell-centered finite difference elliptic equation solver. **PARCEL** can be used to compute a cell-centered finite difference approximation (or equivalently, the lowest-order Raviart-Thomas-Nedelec (RTN_0) mixed finite element approximation) to linear, second-order elliptic equations of the form,

$$\alpha p - \nabla \cdot D \nabla p = f \quad \text{in } \Omega, \tag{1}$$

with appropriate boundary conditions. The domain Ω is a rectangular solid region in $\mathbb{R}^3$. More general geometries can be handled via coordinate transformations; see, for example, [1, 2]. The functions α, D, f and the boundary conditions may all be spatially varying. The coefficient α must be a nonnegative function that may be identically zero. D is a symmetric positive definite matrix, not necessarily diagonal.

Problems in the form of (1) arise in many applications including subsurface flow (see, e.g., [9]), electrostatics and implicit time discretizations of parabolic and hyperbolic equations in which case α is related to the size of the time step. An instance of (1) also arises as a subproblem in the application of the modified method of characteristics to advection-diffusion equations [5].

PARCEL solves (1) using a substructuring domain decomposition method in which the domain Ω is partitioned into a number of non-overlapping subdomains. New unknowns are introduced along the interface between subdomains, and a reduced problem is formulated in terms of these new unknowns. This approach was first proposed by Glowinski and Wheeler in [7] where they refer to this procedure as "Method 2". The reduced interface problem is solved using a preconditioned conjugate gradient routine. The preconditioning options implemented in **PARCEL** include an approximate Jacobi preconditioned conjugate gradient method described in [4] and a state of the art Balancing preconditioner due to Mandel described and analyzed in [8, 3].

PARCEL is a collection of several modules written in Fortran. The only valid entry point into the **PARCEL** package is the subroutine *Parcel3D()*. In order to provide a certain level of portability across computing platforms, **PARCEL** is implemented in a low-overhead communications platform with support for the **PICL**, **PVM**, **NX** and **CMMD** communications libraries. The code was developed on the Intel family of machines, and has been tested on a network of workstations.

[*]AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974-0636

[†]Texas institute for Computational and Applied Mathematices, University of Texas, Austin, Texas 78712

2 Discretization and Solution Algorithm

PARCEL provides a node callable Fortran subroutine *Parcel3D()* that computes the expanded mixed finite element approximation to the following second order scalar elliptic equation,

$$\tilde{\mathbf{u}} = -\nabla p \qquad \text{in } \Omega\,, \tag{2}$$

$$\mathbf{u} = D\tilde{\mathbf{u}} \qquad \text{in } \Omega\,, \tag{3}$$

$$\alpha p + \nabla \cdot \mathbf{u} = f \qquad \text{in } \Omega\,, \tag{4}$$

$$p = q \qquad \text{on } \hat{\Gamma}\,, \tag{5}$$

$$\mathbf{u} \cdot \mathbf{n} = g \qquad \text{on } \Gamma^*. \tag{6}$$

The domain Ω and each subdomain is assumed to be a rectangular solid with a tensor product grid. $\hat{\Gamma}$ and consequently Γ^* are assumed to each be the union of faces of cells on the outer boundary of Ω. Furthermore, $\hat{\Gamma}$ and Γ^* are not assumed to be connected regions.

PARCEL employs a cell–centered finite difference scheme equivalent to the lowest-order Raviart-Thomas-Nedelec expanded mixed method with quadrature to discretize the equation over Ω. Thus, system unknowns are approximations to the scalar variable p at cell centers. Extra unknowns are added along interfaces between subdomains, and the cell center unknowns are eliminated in terms of these interface unknowns. The interface unknowns can be thought of as trace values of p at these interface points. The resulting problem then is to solve a linear system whose solution is an approximation to p along the interfaces between subdomains.

In [7, 4] two substructuring domain decomposition schemes appropriate for mixed finite element discretizations were developed. In the scheme considered here, unknowns corresponding to the common value of p along the interfaces between subdomains are introduced. These values are adjusted iteratively using a conjugate gradient method until the values of $\mathbf{u} \cdot \mathbf{n}$ match on the interface.

Each step of the algorithm proceeds as follows:

1. Choose values of p along subdomain interfaces.

2. Solve problems on each subdomain using the interface values above as Dirichlet boundary data.

3. From the subdomain solutions for p, calculate values of $\mathbf{u}$ and check if values of $\mathbf{u} \cdot \mathbf{n}$ match at the subdomain interfaces.

This defines the basic solution approach.

3 The Balancing Preconditioner

While the interface problem is better conditioned than the original problem over cell-center values of p, further precondtioning may be necessary. A particularly effective preconditioner is Mandel's Balancing preconditioner described and analyzed in [3, 8]. In this section we give a brief overview of the balancing preconditioner implemented in **PARCEL**. The reader is refered to [3, 8] for further information.

The balancing preconditioner uses a "coarse space" which spans the null space of the interface operator S. This coarse space has far fewer degrees of freedom than the original problem. Denote the coarse space as W.

The action of the balancing preconditioner M_{bal}^{-1} on a vector r is computed in the following way:

1. For the first residual of the conjugate gradient procedure, find $w_H \in W$ satisfying

   $$(Sw_H, v_H) = (r, v_H), \quad v_H \in W.$$

   Set

   $$r_{bal} = r - Sw_H.$$

2. Find any solution to the local Neumann problems

   $$S_i \lambda_i = r_{bal_i},$$

   where S_i is the projection of the operator S onto the degrees of freedom associated with subdomain i, and r_{bal_i} is the restriction of r_{bal} to subdomain i.

3. Assemble the λ_i into a λ defined over the entire interface.

4. Find λ_H satisfying

   $$(S(\lambda + \lambda_H), v_H) = (r, v_H), \quad v_H \in W.$$

5. Set

   $$M_{bal}^{-1} r = \lambda + \lambda_H.$$

Thus, considering both the linear system solution and the application of the preconditioner, each preconditioned conjugate gradient iteration requires:

1. The solution of a Dirichlet problem on each subdomain arising from the substructuring domain decomposition algorithm.

2. The solution of a Neumann problem on each subdomain arising from the balancing preconditioner.

3. A coarse solve, arising from the preconditioning step, which involves communication between processors.

Data and work for the linear system solution are distributed over the processors. For ease of computation, we assume that each processor holds data for exactly one subdomain. The processors then can each factor and solve the local Dirichlet and Neumann subdomain problems independently. Direct methods are used for these solves. The coarse problem matrix is formed and factored before the conjugate gradient iteration begins. The factorization and solves with this matrix are also done with direct methods. Thus, the subdomain problems are all handled locally while the coarse solve is done globally and involves some communication overhead.

4 The *Parcel3D()* subroutine

In this section the calling sequence of *Parcel3D()* is discussed. This is the only valid entry into the **PARCEL** package. A call to *Parcel3D()* has the following form:

```
 call parcel3D(majv,minv,
$     n1,n2,n3,x1,x2,x3,l1,l2,l3,pres,vel1,vel2,vel3,
$     alpha,D,Dtype,f,bcType_in,nBCRegions_in,bcRegion1,bcRegion2,
$     bcRegion3,bc1,bc2,bc3,mynbr,icube,rparam,iparam,idecomp,
$     SDparam,rwk,len,iwk,ilen)
```

We will now describe the subroutine arguments. The arguments serve three purposes: describe the local partial differential equation, describe the connectivity of the decomposition, and provide tolerances and other parameters to the interface and subdomain problems.

Since **PARCEL** is a node–callable library, the code assumes that the global problem grid and relevant problem information has been distributed to the nodes. Thus, all parameters to the *Parcel3D* subroutine are for the SUBDOMAIN problem for the node calling the routine and not for the GLOBAL problem.

- majv: Integer [INPUT] In an effort to make sure that the user is actually calling the release of the package he thinks he is, the user must provide the correct major release number for the **PARCEL** package. This document provides documentation for Major Release 1. Hence, majv should be set equal to 1. A change in Major Release number may signify a major change in the argument list of *parcel3D*; hence, a more current copy of the documentation should be used. If a mismatch of Major Release number occurs, then *parcel3D* aborts.

- minv: Integer [INPUT] Like the Major Release number, the Minor Release number is used to make sure that the user is calling the right version of the package. If a Minor release number mismatch occurs, execution continues and a warning message is given. At the time of writing this documentation the Minor Release number was 4.

The majority of parameters are concerned with the specification of the differential problem to be solved.

- n1: Integer [INPUT] The number of cells in the first coordinate axis of the *subdomain problem.* Each subdomain is rectangular with a tensor product grid.

- n2: Integer [INPUT] The number of cells in the second coordinate axis of the *subdomain problem.*

- n3: Integer [INPUT] The number of cells in the third coordinate axis of the *subdomain problem.*

- x1(0:n1): Real*8 Vector [INPUT] The mesh points in the first coordinate direction. It is assumed that x1(0) < x1(n1).

- x2(0:n2): Real*8 Vector [INPUT] The mesh points in the second coordinate direction. It is assumed that x2(0) < x2(n2).

- x3(0:n3): Real*8 Vector [INPUT] The mesh points in the third coordinate direction. It is assumed that x3(0) < x3(n3).

- l1(n2,n3,2): Real*8 Array [INPUT/OUTPUT] On input, the guess for the inter-subdomain multipliers (denoted λ in [4]) for the faces with normals in the x1-direction. l1, l2, and l3 are approximations to p along the interfaces of subdomains. In particular l1(*,*,1) corresponds to the face $x_1 \equiv$ x1(0) and l1(*,*,2) to the face $x_1 \equiv$ x1(n1). When a subdomain does not share a face, i.e. the face is part of the outer boundary, those unknowns should be set to zero. On output, l1 contains the converged solution for the inter-element multipliers on the corresponding faces.

- l2(n1,n3,2): Real*8 Array [INPUT/OUTPUT] On input, the guess for the inter-subdomain multipliers (denoted λ in [4]) for the faces with normals in the x2-direction. In particular l2(*,*,1) corresponds to the face $x_2 \equiv$ x2(0) and l2(*,*,2) to the face $x_2 \equiv$ x2(n2). When a subdomain does not share a face, i.e. the face is part of the outer boundary, those unknowns should be set to zero. On output, l2 contains the converged solution for the inter-element multipliers on the corresponding faces.

- l3(n1,n2,2): Real*8 Array [INPUT/OUTPUT] On input, the guess for the inter-subdomain multipliers (denoted λ in [4]) for the faces with normals in the x3-direction. In particular l3(*,*,1) corresponds to the face $x_3 \equiv$ x3(0) and l3(*,*,2) to the face $x_3 \equiv$ x3(n3). When a subdomain does not share a face, i.e. the face is part of the outer boundary, those unknowns should be set to zero. On output, l3 contains the converged solution for the inter-element multipliers on the corresponding faces.

- pres(n1,n2,n3): Real*8 Array [INPUT/OUTPUT] On input, this array holds the initial guess for the subdomain cell centered pressures. On return, the array holds an approximation to the pressure that solves (2)–(6). pres(i1,i2,i3) is an approximation to p at the center of cell (i1,i2,i3), i.e. at the point ((x1(i1)+x1(i1-1))/2 , (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2).

- vel1(0:n1,n2,n3): Real*8 Array [OUTPUT] This array holds the x1-component mixed method velocity for this subdomain.

- vel2(n1,0:n2,n3): Real*8 Array [OUTPUT] This array holds the x2-component mixed method velocity for this subdomain.

- vel3(n1,n2,0:n3): Real*8 Array [OUTPUT] This array holds the x3-component mixed method velocity for this subdomain.

- alpha(n1,n2,n3): Real*8 Array [INPUT] The lower order term α in equations (2)–(6). The entries in alpha should be non-negative. alpha(i1,i2,i3) is either the value of α at the center of cell (i1,i2,i3), i.e. at the point ((x1(i1)+x1(i1-1))/2 , (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2), or the cell average of α on cell (i1,i2,i3).

- D : ?? [INPUT] The datum D contains information used to describe the tensor D in (3). The most natural way to specify this information seems to be application specific. After some debate, the specific contents of D are decided to be left unspecified. D is passed to several routines that construct the local subdomain matrices and estimate the diagonal of local contributions to the Schur complements. Currently the following options are supported. D may be one of two types of vectors or it may be a function name used to calculate the values of the tensor. If D is of the first vector type, it should be a real*8 long vector of length ((n1+1)*n2*n3 + n1*(n2+1)*n3 + n1*n2*(n3+1)) that is the concatenation of the x1, x2 and x3 edge transmissibilities. Note that in this case D is assumed to be a diagonal tensor. If D is of the second vector type, it should be a real*8 long vector of length 6*n1*n2*n3 that is the concatenation of the values at the cell centers for each of the 6 possible entries of D (assuming D is a symmetric tensor). Note that in this case the ordering of the 6 entries is $D_{11}, D_{12}, D_{13}, D_{22}, D_{23}$, and D_{33}. If D is a function, D is a real*8 function of the form:

```
real*8 function userD(i,j,i1,i2,i3)
integer i,j,i1,i2,i3
```

  where userD returns the i,j-entry of the tensor D at the center of local cell (i1,i2,i3).

- Dtype: Integer [INPUT] Dtype describes the type of representation of D that is used above. If Dtype=1, then it is stored as a long vector of edge transmissibilities, if Dtype=2, then it is a user supplied function taking arguments as described above. If Dtype=3, then it is stored as a long vector of cell centered values. Symbolic names are defined for these types in the file ../Include/defines.h

- f(n1,n2,n3): Real*8 Array [INPUT] f is the right hand side source term f in (2)–(6). The average value of f in the cell is a reasonable choice for its discretization.

- bcType_in(nBCRegions_in): Integer Array [INPUT] Contains the type of boundary condition for each boundary region of the GLOBAL problem. The types are given symbolic names in ../Include/BCtypes.h, and the values are: 1 if the region is Neumann, 2 if the region is Dirichlet and 3 if the region is No Flow (i.e. Neumann with 0 flux).

- nBCRegions_in: Integer [INPUT] The total number of boundary condition regions for the global problem.

- bcRegion1(n2,n3,2): Integer Array [INPUT] This array contains the boundary region number to which each boundary cell with normal in the x1-direction belongs. The third index is 1 if the cell has a negative outward normal and 2 if it has a positive outward normal.

- bcRegion2(n1,n3,2): Integer Array [INPUT] This array contains the boundary region number to which each boundary cell with normal in the x2-direction belongs.

- bcRegion3(n1,n2,2): Integer Array [INPUT] This array contains the boundary region number to which each boundary cell with normal in the x3-direction belongs.

- bc1(n2,n3,2): Real*8 Array [INPUT] This array contains the value of the subdomain boundary conditions for boundary cells with normals in the x1-direction. If a face is internal, bc(*,*,iface) should contain 0's. If a face is on the outer boundary and Dirichlet data is imposed on cell (i1,i2) of that face (indicated by bcType_in = 2), then bc(i1,i2,iface) should contain the value of p at (x1(0), (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2) if iface is 1, and the value of p at (x1(n1), (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2) if iface is 2. For No Flow boundary conditions bc1(*,*,iface) should contain 0's. And for Neumann conditions, bc1(*,*,iface) should contain the value of the VOLUMETRIC flux $-D\nabla p \cdot \mathbf{n}$ times (x2(i2)-x2(i2-1))*(x3(i3)-x3(i3-1)) at (x1(0), (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2) if iface is 1, and the same quantity at the point (x1(n1), (x2(i2)+x2(i2-1))/2, (x3(i3)+x3(i3-1))/2) if iface is 2. Here $\mathbf{n}$ is the outward normal.

- bc2(n1,n3,2): Real*8 Array [INPUT] This array contains the boundary condition values for subdomain faces with normals in the x2-direction. See bc1 for a complete description.

- bc3(n1,n2,2): Real*8 Array [INPUT] This array contains the boundary condition values for subdomain faces with normals in the x3-direction. See bc1 for a complete description.

The next few arguments are vectors that encapsulate some properties of the domain decomposition or some parameters describing the iterations. They are grouped together to try to provide a partitioning of certain aspects of the code.

- mynbr(3,2): Integer Array [INPUT] This variable describes the local connectivity of the subdomains. The array contains the node numbers of the neighboring nodes. Mynbr(idir,iface) should be the node number of the neighbor sharing the face with a normal in the positive "idir"-direction if iface=2, or negative "idir"-direction if iface =1. If a node has no neighbor node for a given face, mynbr(idir,iface) MUST be set to -1.

- icube(3): Integer Vector [INPUT] Describes some of the "cube information".
  - [icube(1)=MYNOD] Node number.
  - [icube(3)=CUBESIZE] Number of processors allocated.
- rparam(12): Real*8 Vector [INPUT/OUTPUT] Real parameters for the interface solver.
  - [rparam(1)=epsIF] Relative reduction required for interface problem. (INPUT)
  - [rparam(2)=tolIF] Absolute tolerance required to solve interface problem. (INPUT)
  - [rparam(3:12)] = times(1:10) On output returns the time to execute certain internal sections of *parcel3D()*. (OUTPUT)
- iparam(8): Integer Vector [INPUT/OUTPUT] Integer parameters for the interface solver.
  - [iparam(1)=outUnit] Output unit number. 6 is standard output. (INPUT)
  - [iparam(2)=itest] Test problem number. Not really used currently. Will be used in output later on. (INPUT)
  - [iparam(3)=iverb] Verbosity. Controls the amount of output to outUnit. iverb = 5 gives interface progress, iverb $\geq$ 15 gives subdomain solve output as well. See Table 2 for more information about the available output. (INPUT)
  - [iparam(4)=idebug] Debug level. (INPUT)
  - [iparam(5)=ising] 0 if global problem is well posed, 1 if problem is singular (full Neumann with $\alpha = 0$) with null space equal to the constant functions. (INPUT)
  - [iparam(6)=PCstyle] The type of preconditioning for the interface problems. 0 is no preconditioning, 1 and 2 are preconditioning by the value of $\mathbf{n}^t D \mathbf{n}$ restricted to the interface and 5 is Balancing preconditioning. (INPUT)
  - [iparam(7)=niterIF] The maximum number of conjugate gradient iterations to perform on the interface problem. (INPUT)
  - [iparam(8)=istat] Status of the interface problem. istat < 0 indicates failure to converge. (OUTPUT)
- idecomp(3): Integer Vector [INPUT] These parameters describe the decomposition. Since the decompositions are thus far just tensor product rectangular, three integers suffice: ndiv1, ndiv2, ndiv3, the number of divisions of the global domain along the 3 coordinate axes.
- SDparam(5): Real*8 Vector [INPUT] These parameters are passed to the subdomain solvers. Currently, a memory intensive "exact solver" using a Cholesky factorization of the local stiffness matrix is used to solve the subdomain problems. Symbolic names are defined in Include/sdsolver.h.

  For the exact solver, one simply sets

  - [SDparam(1)=3] for the diagonal tensor case, or
  - [SDparam(1)=5] for the full tensor case.

The last arguments are the work space vectors. **PARCEL** allocates very little static memory. Most of the allocations are made from the work vectors using the STACK memory utilities found in Memory/memory.f.

- rwk(len): Real*8 Vector [INPUT] Real work vector.
- len: Integer [INPUT/OUTPUT] On input, the length of the rwk work vector. On output, the maximum length of rwk used.
- iwk(ilen): Integer Vector [INPUT] Integer work vector.
- ilen: Integer [INPUT/OUTPUT] On input, the length of the iwk work vector. On output, the maximum length of iwk used.

5 Return Codes

Errors in **PARCEL** fall into two categories, catastrophic and recoverable. A catastrophic error is an error from which computation may not proceed. A recoverable error is a state in which certain criteria may be violated, but the calculation may proceed. For example, if a node does not have enough memory to allocate all storage, that is a catastrophic error. If a given subdomain fails to converge in the prescribed number of iterations, the computation may be suspect, but the iteration on the interface problem may continue. In the event of a catastrophic error, a message is sent to the screen and the program aborts. It may be necessary for the user to then kill and release the cube manually (if running on a hypercube) or to reset the PVM process. This may be necessary since some nodes may not experience a catastrophic error and those nodes would block and hang. Recoverable errors are reported to the user, but computation continues.

Catastrophic errors result in the node on which the error occurred calling the routine:

```
      subroutine Parcelcrash(msg,errorcode)
c
      integer errorcode
      character*(*) msg
```

This routine must be supplied by the user and should take appropriate action to abort the computation. The two arguments are an error code and a descriptive message concerning the error.

For most modules of **PARCEL**, a return code greater than or equal to zero signifies success. A return code less than zero signifies some degree of failure. Catastrophic errors are usually reported by a return value of less than or equal to -10.

Table 1 lists the error and return codes that may be returned.

6 Compiling and Running PARCEL

This section discusses how to compile and run the **PARCEL** code and test program *parceltst*. We assume the source code has been installed.

After installing **PARCEL**, change to the ParcelDist directory. You should see the following files and directories

```
Bcfdnf          LOGFILE         Parcel
Com             Makefile        Test
Doc             Memory
Include         Packages
```

The contents of the directories are:

Bcfdnf Direct Cholesky subdomain solvers for diagonal and full tensor cases.

Com Definitions of the generic communications routines for various computing platforms

Doc This documentation

Include Directory for shared include files

LOGFILE File describing changes made to the code

Makefile Makefile for **PARCEL**

Memory Stack memory allocation routines

Packages Routines from Blas, Linpack and Eispack

Parcel The **PARCEL** source code

Test Routines for a test program that uses **PARCEL**

A full test program and a corresponding suite of test problems is contained in the Test subdirectory. The next two paragraghs discuss how to compile and execute **PARCEL** with this program driving it. A rather elaborate suite of sample test problems is provided in the directory **Test**.

To construct an executable file using the **PICL**communication library, do the following. The **PICL**communication libraries and a **PICL**users' guide which contains instructions on compiling and creating these libraries can be obtained from the server netlib@ornl.gov. Once the libraries have been installed, assign the path of the **PICL**node library to the variable PICLLIB in the Test directory Makefile. From the ParcelDist directory, type "make picl". This command will create all the libraries needed for **PARCEL** and the executable file for the test program. This executable is called "parceltst" and will be in the Test subdirectory. There is also a shell script, written for an Intel i860 computers, called "runit860" in the Test subdirectory. This script allocates a cube, runs the test program and releases the cube. The script takes the size of the cube (number of subdomains) as its argument. In order to run *parceltst* on the Intel i860, the number of nodes must be a power of two or the cube will hang. For the sample input given in Appendix A, the command "runit860 8" from the Test subdirectory will execute the test program.

Currently, **PARCEL** only runs with the SUN4 implementation of **PVM**. To construct an executable file using the **PVM**communication library, do the following. Obtain **PVM**version 3.0 or higher from the server netlib@ornl.gov or form some other source. Following the documentation in the **PVM**users guide [6], compile the appropriate libraries and executables necessary for the SUN4 implementation with dynamic process groups. These files include: libpvm.a, libfpvm.a, libgpvm.a, pvmgs, pvm_gstat and pvmd. Assign the location of your pvm3 directory to the name PVMDIR and specify the locations of the above three libraries by modifying the PVMLIBS variable in the Makefile contained within the Test directory. **PARCEL** assumes that there is a SUN4 subdirectory within your PVMLIBS directory, and it will look there for the above libraries. If these libraries are found somewhere else, modify the variables LIBFPVM, LIBGPVM, and LIBPVM in the Makefile within the Test directory to give their location. After specifying the library locations, typing "make sun4" will make the libraries for **PARCEL** and the executable test program "parceltst." In accordance with **PVM**convention, the executable test program will be moved to the pvm3/bin/SUN4 directory. All that remains is to put a copy of the input file into the pvm3/bin/SUN4 directory.

Also from the base directory, one can execute "make clean" to remove all the intermediate files such as "*.o" files.

7 Some Notes on the Communication Library

PARCEL has been developed on the Intel family of machines using the **PICL**library to provide communication primitives. It has subsequently been ported to a set of generic communication routines which support the **PICL**, **PVM**, **NX** and **CMMD** communication libraries. The choice of communication libraries is made when **PARCEL** is compiled. Therefore, to use **PARCEL** in its present state, the user must write the driver program using the appropriate communication library or using the general communication library supplied with **PARCEL**.

The rest of the section discusses the generic communication interface found in the Com directory and modifications necessary for running **PARCEL** on a **PVM**with architectures not currently supported.

The files in the Com directory contain definitions of these generic routines in terms of **PICL**, **PVM**, **NX** and **CMMD**. To add a new communications paradigm, one would just have to write a new file containing definitions of these generic routines in terms of the new paradigm.

Currently, all libraries for **PARCEL** are compiled and created using recursive Makefiles. The Makefile in the base directory sets a variable, ARCH, to either SUN4 or PICL, then switches to each subdirectory and calls the corresponding Makefile. Each of these Makefiles sets the appropriate compiler flags and libraries for the architecture and calls itself in order to create the library. To add another architecture to this system, one would need to modify the base directory Makefile to include a value of ARCH for the new architecture. The best choice is the name given in the **PVM**documentation as that will allow for easy access to the **PVM**libraries. Then, within each Makefile in directories *Bcfdnf, Com, Memory, Packages, Packages/linpack, Parcel* and *Test*, labels must be added corresponding to the new ARCH extension and specifying appropriate compiler options.

Appendix A: Test Program Input file

This section describes the input file used in the test program **parceltst** found in the Test directory.

The input file name "input" is hard coded in the driver source code (*parceltst.f*). A sample input file is contained below.

```
   11                          Test Problem Number
   5                           Verbosity of output
   0                           Debug Level
   32                          N1
   24                          N2
   16                          N3
   4                           NDIV1
   2                           NDIV2
   1                           NDIV3
   5                           PC Style: 0 = I, 1=Avg. 2=Har. 4=BP(0) 5=BAL
  0.                           Robin B.C. factor
 100                           Maximum number of iterations to solve IF prob.
1.0e-6                         Relative resid. reduction for Interface
1.0e-20                        Absolute resid.  for Interface
 100                           Max iterations in Subdomain Problem
   6                           Number of orthog. directions
1.0e-9                         Relative tolerance for subdomains
 0.0                           Absolute tolerance for subdomains
   3                           Solver: 3 = ex. diag., 5 = ex. full
uniform                        File: BC regions (or 'uniform')
   6                           Number of BC regions
   2                           BC for region 1
   2                           BC for region 2
   3                           BC for region 3
   3                           BC for region 4
   3                           BC for region 5
   3                           BC for region 6
<uniform BC-------------------------------------------------------
   2                           BC for {x1=0} edge  Boundary Conditions
   2                           BC for {x1=1} edge      1 - Neumann,
   1                           BC for {x2=0} edge      2 - Dirchlet,
   1                           BC for {x2=1} edge      3 - No Flow
   2                           BC for {x3=0} edge      4 - Robin BC
   2                           BC for {x3=1} edge
   1                           Grid Type:  1 - Uniform Grid 0 - Nonuniform
   0.                          x1(0)
   1.                          x1(n1)
   0.                          x2(0)
   1.                          x2(n2)
   0.                          x3(0)
   1.                          x3(n3)
```

These inputs select test problem number 11, described in the routines in *user.f*. The output level is 5 and the debug level is 0. These two parameters control the amount of output written to the screen, cf. Table 2.

The global problem will consist of a grid that is 32x24x16, divided into 8 subdomains in a 4x2x1 configuration. Hence each subdomain will be 8x12x16.

The balancing preconditioner is used on the interface problem. Iteration on the interface problem will continue until the norm of the relative residual is reduced by 1.0e-6, the norm of the absolute residual becomes less than 1.0e-20, or 100 iterations are completed.

The next entries in the input file specify parameters for the solution of the subdomain problems if an iterative method is used. These options are currently not used.

The solver type is 3 indicating an exact solver for a diagonal tensor coefficient local problem. An exact solver for the full tensor coefficient local problem is also available.

The boundary conditions are uniform, i.e. each face of the global boundary has the same boundary condition. For this test problem the domain faces x1=0, x1=1, x3=0 and x3=1 have Dirichlet boundary conditions specified, whereas the faces x2=0 and x2=1 have Neumann boundary conditions specified. For uniform boundary conditions, the number of boundary condition regions and the type for each of these regions is irrelevant. For nonuniform conditions a boundary condition file must be specified (see Appendix B) as well as the number of regions and the type of condition in each.

The last entries specify the grids. If the grid is chosen uniform (as in this case), only the end points of the intervals need to be specified. For nonuniform grids, instead of $x_1(0)$ and $x_1(n_1)$, the input file would contain $x_1(0)$, $x_1(1)$, $x_1(2)$, $\ldots$, $x_1(n_1)$ followed by like grids for x_2 and x_3.

Several variables control the amount of output the user sees. While it is useful to observe such data to gain insight into the calculations, the user should specify the minimum amount of output that still gives the user peace of mind. The output of data to the screen is a very time costly procedure on most machines. Even a relatively small amount of output can double run time! Table 2 lists the additional output one obtains by increasing the verbosity variable. There are several jumps in the verbosity values to provide room for expansion and customization.

Appendix B: Setting Up Boundary Conditions for the Test Program

This section exlains conventions for setting up boundary conditions for the *parceltst* program. If the boundary conditions are uniform, one boundary condition is specified for each face of the global domain.

More generally, the outer boundary is divided into regions where each region has the same type of boundary condition. The regions are specified in a file (with a name given in the input file) in the following format.

```
   n1,n2,n3 (global problem size)
c
c left face
c
      ((bcRegion(j,k),j = 1,n2),k = 1,n3)
c
c right face
c
      ((bcRegion(j,k),j = 1,n2),k = 1,n3)
c
c front face
c
      ((bcRegion(i,k),i = 1,n1),k = 1,n3)
c
c back face
c
      ((bcRegion(i,k),i = 1,n1),k = 1,n3)
c
c bottom face
c
      ((bcRegion(i,j),i = 1,n1),j = 1,n2)
c
c top face
c
      ((bcRegion(i,j),i = 1,n1),j = 1,n2)
    end
```

Here, bcRegion is the array into which the values of the region numbers will be read for each boundary cell of the global domain. For instance, the left face has n2 x n3 cell faces. The region to which each cell belongs is specified for that face. The next face will be the right face, and so on. The boundary condition type for each region is given in the input file.

A preprocessing program called "rdistr" contained in the Test directory must be run prior to running *parceltst*. This program reads the input and the boundary condition regions files, and creates files called innode???? (???? = 0000-9999 is the node number) for all processors. These files contain the boundary condition regions data and are read in **PARCEL** by the processors.

Appendix C: Test Program Sample Output

The following is the output from a single run with the above input file.

```
 PARCEL (Parallel Cell-Centered Elliptic Solver)
 PARCEL Release V1.4
 (  $Revision:$1.1$$  $Date:  1994/08/01  21:47:54  $  )
 Test  Problem  Number  :   11  Verbosity:    5  Debug:    0
 input n1 n2 n3
        32        24        16
 input ndiv1 ndiv2 ndiv3
         4         2         1
 PCstyle:           5
 RobinR:   0.0000000000000000E+000
 Max. Interface Its. eps_if tol_if
  100      0.10000000E-05      0.10000000E-19
 niter northog  eps_sd tol_sd
  100    6      0.10000000E-08      0.00000000E+00
 solverType:           3
 BCs:           2           2           1           1           2           2
 X1:   0.0000000000000000E+000   1.000000000000000
 X2:   0.0000000000000000E+000   1.000000000000000
 X3:   0.0000000000000000E+000   1.000000000000000
 Asymmetry is:  1.9984014443252818E-015
 Balance matrix is structurally singular           0
 Continuing with eigen-decomposition
 Setup Time:    9.884000     (secs)
 Residual Time:   0.1860000     (secs)
CGIter:     0  Residual Norm:  0.105E+01 ( 0.240E+00)
CGIter:     1  Residual Norm:  0.336E+00 ( 0.320E+00)
CGIter:     2  Residual Norm:  0.936E-01 ( 0.892E-01)
CGIter:     3  Residual Norm:  0.231E-01 ( 0.220E-01)
CGIter:     4  Residual Norm:  0.582E-02 ( 0.555E-02)
CGIter:     5  Residual Norm:  0.117E-02 ( 0.111E-02)
CGIter:     6  Residual Norm:  0.260E-03 ( 0.247E-03)
CGIter:     7  Residual Norm:  0.526E-04 ( 0.501E-04)
CGIter:     8  Residual Norm:  0.601E-05 ( 0.572E-05)
CGIter:     9  Residual Norm:  0.375E-06 ( 0.357E-06)
 Final:        1  1   0.105D+01   0.375D-06    0  0  0  0  0  0  0  0
 Norms:           1   0.105D+01   0.375D-06   0.280D+07   0.357D-06
 Soln Time:    4.153000     (secs)
 l2 error =  2.5847945692002616E-004
 velocity l2 error =  3.1397510214526496E-003
 velocity L-inf error =  1.7078651191508504E-002
Node    0 returns    0 Time:   15.943  14.507    9.88   8.00   4.18   4.42
 Max Total Time : Iter: Condition : Status
   14.50700     8   0.273554E+01  0
```

We first see an identification message giving the version number and revision date. The next section of output just echos the reading of the input file. The next line starts output from the node. Asymmetry refers to the degree of asymmetry in the balancing matrix. The time to build the preconditioner and factor the local problem matrices was then reported in the Setup Time. The Residual Time gives the time taken in putting the problem in residual form, see [4]. The next five lines give information on the conjugate gradient iterations. The solution time for node 0 is then given. We next see various errors of the pressure and velocity solutions. Lastly, the solution time for the slowest node is given along with the number of conjugate gradient iterations and the condition number of the problem.

Appendix D: Internal Distribution Notes

If you have an account on the Texas Institute for Computational and Applied Mathematics network at the University of Texas, the best way to obtain the **PARCEL** code is to get in touch with Ivan Yotov (yotov@ticam.utexas.edu). The files are controlled by the revision control system which handles automatic updating of new versions. Your version of **PARCEL** will be able to take advantage of this updating.

To install **PARCEL**, you will need the shell script "installparcel". When executed from your root directory, this script will set up the necessary structure for you to use the latest version of **PARCEL** and fetch this version for you. The only thing left will be for you to compile the appropriate version.

References

[1] T. Arbogast, M. F. Wheeler, and I. Yotov, *Logically rectangular mixed methods for groundwater flow and transport on general geometry*, Dept. Comp. Appl. Math. TR94-03, Rice University, 1994.

[2] ——, *Mixed finite elements for elliptic problems with tensor coefficients as cell-centered finite differences*, Dept. Comp. Appl. Math. TR95-06, Rice University, Houston, TX 77251, Mar. 1995. To appear SIAM J. Numer. Anal., 1997, vol. 34.

[3] L. C. Cowsar, J. Mandel, and M. F. Wheeler, *Balancing domain decomposition for mixed finite elements*, Mathematics of Computation, 64 (1995), pp. 989–1016.

[4] L. C. Cowsar and M. F. Wheeler, *Parallel domain decomposition method for mixed finite elements for elliptic partial differential equations*, in Proceedings of the Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations, R. Glowinski et al., eds., SIAM, 1991.

[5] J. Douglas, Jr and T. F. Russell, *Numerical methods for convection-dominated diffusion problems based on combining the method of characteristics with finite element or finite ifference procedures*, SIAM J. Numer. Anal., 19 (1982), pp. 871–885.

[6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *Pvm 3.0 user's guide and reference manual*, Tech. Rep. ORNL/TM-12187, Oak Ridge National Laboratory, Feb. 1993.

[7] R. Glowinski and M. F. Wheeler, *Domain decomposition and mixed finite element methods for elliptic problems*, in Proceedings of the First International Symposium on Domain Decomposition Methods for Partial Differential Equations, R. Glowinski et al., eds., SIAM, Jan. 1987, pp. 144–172.

[8] J. Mandel, *Balancing domain decomposition*, Comm. on Applied Num. Methods, 9 (1993), pp. 233–241.

[9] T. F. Russell and M. F. Wheeler, *Finite element and finite difference methods for continuous flows in porous media*, in Mathematics of reservoir simulation, R. E. Ewing, ed., SIAM, Philadelphia, 1983, ch. II, pp. 35–106.

Return Code	*Meaning*	*Solution*
1	Convergence in the interface problem via the absolute tolerance criteria	
0	Convergence in the interface problem via the relative residual criteria	
-1	Maximum number of interface iterations exceeded	Results suspect
-2	Too many conjugate gradient iterations	Results suspect
-3	Nested Factorization failed	Results suspect
-10	Problem too large	Increase *inmax* or decrease local problem size
-12	Not enough 1D space on a node	Increase *inmax* in **size.h**
-15	Memory allocation error	
-16	Not enough work space	Increase *inmax* or *idim3*
-17	Grids not nested	Improper local grid refinement
-30	Generic error	To be categorized soon
-31	Major Version number incorrect	Get updated documentation and correct variable majv in call to parcel3D()
-32	Cube too large	Reset MAXCUBE in "Include/msgtypes.h" and recompile parcel3D()
-33	Unknown Interface Matching Method	
-34	Balancing Preconditioner is not positive definite	Use other PC, use exact solves
-35	Solver type not supported	
-36	Specified more than the max. number of boundary condition regions	Adjust BCRegions in size.h
-37	Unknown boundary condition type	
-38	Error reading boundary condition types	
-39	Inconsistent boundary conditions	

Table 1: Return Codes

Verbosity	*Output*
0	Fatal error messages only
1	Run time data
5	Status message per conjugate gradient iteration
6	Eigenvalue estimate from conjugate gradients
15	Subdomain iterations

Table 2: Verbosity and Output